IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

APPLICATION FOR U.S. LETTERS PATENT

Title:

METHOD AND APPARATUS FOR SHARING STANDARD TEMPLATE LIBRARY OBJECTS AMONG PROCESSES

Inventor:

Vijapurapu V. Anantharao
2275 S. Bascom Avenue, #1706
Campbell, CA 95088

Citizenship: Indian

METHOD AND APPARATUS FOR SHARING STANDARD TEMPLATE LIBRARY OBJECTS AMONG PROCESSES

RELATED APPLICATIONS

[0001] The present application is a continuation of co-pending and commonly assigned U.S. Patent Application Serial No. 09/568,562, filed May 9, 2000, and entitled "METHOD AND APPARATUS FOR SHARING STANDARD TEMPLATE LIBRARY OBJECTS AMONG PROCESSES," the disclosure of which is hereby incorporated herein by reference.

COPYRIGHT NOTICE

[0002] A portion of the disclosure of this patent document contains material which is subject to copyright protection. The copyright owner has no objection to the facsimile reproduction by anyone of the patent document or the patent disclosure, as it appears in the Patent and Trademark Office patent file or records, but otherwise reserves all copyright rights whatsoever. The copyright owner does not hereby waive any of its rights to have this patent document maintained in secrecy, including without limitation its rights pursuant to 37 C.F.R. §1.14.

TECHNICAL FIELD

[0003] This invention relates to memory management in a computer system, and more particularly to sharing of lookup data by loading lookup data into standard template library objects (STL objects) in a shared memory.

BACKGROUND

[0004] Modern data processing systems that support several processes simultaneously are defined as multiprocessing systems. More specifically, a process can be defined as an address space, a thread of control operating within that address space, and a set of required system resources for supporting the execution of a program. Before execution, the data needed for program execution and on which the program operates is loaded into the address space. In a multiprocessing system, several processes may execute and share data; some of the data used in different processes are the same for subsequent processes, and are termed "shared data."

[0005] In multiprocessing systems, a plurality of processes may execute simultaneously for higher performance. The processes may be interconnected by a system bus through which they communicate among themselves and with commonly shared resources, such as input/output units and working memories. Shared memory is a reserved memory area to which all of the sharing processes have access; different access methods allow the processes to use the shared memory with different performance characteristics.

[0006] Sharing of data structures between processes, specially the sharing of Standard Template Library object (STL objects), has always been a desirable functionality. The Standard Template Library (STL) comprises a set of C++ generic data structures and algorithms, and the STL provides reusable, interchangeable components adaptable to many different uses in software development environment, some of which are described by G. Bowden Wise in *An Overview of the Standard Template Library*, available at http://www.cs.rpi.edu/~wiseb/xrds/ovp2-3b.html. This article is hereby incorporated by reference in its entirety.

[0007] Conventional operating systems such as Windows NT and other operating systems such as UNIX provide shared memory. NT provides shared memory in the form of memory-mapped files. However, the shared memory segment is not necessarily loaded in at the same address by default in different processes or in the same program. Specifically, the MapMemoryFileEx Application Program Interface (API) function is used to request NT to load the shared memory segment at a particular address. It is of very little relevance whether the memory segment is loaded in the same address for offset-based data structures like an array, because, even if the same data structure is loaded at different addresses in different processes, the computations are done based on the offsets. It is of great importance, however, whether the memory segment is loaded in the same address for pointer-based data structures, where an address in one process is not correlated to an address in another process.

SUMMARY

[0008] STL objects are pointer-based data structures. Addressing the deficiency in the traditional file-based approach, this invention provides a technique to load STL objects to allow sharing of the STL objects by a plurality of processes. In one embodiment, this invention implements sharing of any number of STL objects between processes and threads using New Technology's (NT's) Section Objects (or shared memory) and special STL allocator function. Data which are needed by more than one process are stored in user STL objects, and the user STL objects are placed in the same memory segment. By attaching the shared memory segment, multiple processes may have access to the stored data.

[0009] In one aspect, the invention features a computer-implemented method or an apparatus comprising a computer-readable medium having a plurality of sequences of instructions stored thereon for sharing data structures among processes by using a shared memory segment. The method includes creating the shared memory segment, anchoring a system STL (Standard Template Library) map in the shared memory segment, receiving a user STL object, obtaining an address for the user STL object, and inserting the user STL object into the system map.

[0010] In another aspect, the invention is directed to a method or an apparatus comprising a computer-readable medium having a plurality of sequences of instructions stored thereon for conducting garbage collection by an allocator, the allocator capable of allocating storage for a user STL object in a shared memory segment common to a plurality of processes. The method includes creating a first map containing a plurality of nodes representing a plurality of free blocks, each block represented in the first map by a node denoting a size and an address of the free block, responding to an allocation request, and responding to a deletion request by entering a size and an address of a deleted block into the first map.

[0011] In yet another aspect, the invention is directed to a computer system implementing a data-sharing scheme capable of supporting a shared memory segment common to a plurality of processes, the computer having a processor for executing instructions, a memory for storing instructions, the computer system. The computer system includes the shared memory segment, a system STL map anchored in the shared memory segment, and a plurality of user

STL objects, wherein each object is named and is seated in the STL map by inserting a name and an address of the object into the system STL map.

[0012] Various implementations of the invention may include one or more of the following features. The user STL object may have a name. The shared memory segment may be named and may comprise approximately lOMB of memory. The user STL object may comprise created or existing user STL object. The obtaining step may consist of returning the address of the user STL object if the name of the user STL object is in the system STL map. The obtaining step may consist of returning a next allocation address from the shared memory segment if the name of the user STL object is not in the system STL map.

[0013] The method may consist of updating the next allocation address by increasing the next allocation address by an allocated size of the user STL object. The inserting step may consist of entering the name and the address of the user STL object into the system STL map.

[0014] Further the method may consist of requesting storage from an allocator, wherein the allocator allocates memory from the shared memory segment to store at least one node of the user STL object and data added to the user STL object. The data may comprise an entity capable of supporting a single writer and a plurality of readers. An access to the address of the next allocation may be synchronized. The allocator may perform garbage collection. The processes may be threaded.

[0015] Implementation of the invention may include one or more of the following features. The responding to an allocation request step may return an address of a free block in the plurality of free blocks if the allocation request is for a storage size equal to or smaller than the free block. The responding to an allocation request step may allocate memory from the shared memory segment. It may provide a snapshot of the shared memory segment. The snapshot may include a total size of the shared memory segment. The first map may be available to only one process of the plurality of processes and the one process may be a writer process. The shared memory segment may be named and may consist of about lOMB of memory. Addresses of the plurality of user STL objects may comprise addresses of the STL objects existing in the system STL map, and may comprise next allocation addresses.

[0016] An allocator may manage a garbage collection process responding to allocation requests and deletion requests. The allocator may allocate memory from the shared memory segment to store at least one node and data contained in each user STL object. At least one node may contain pointers to accounting information. A first map may contain a first set of nodes denoting sizes and addresses of a plurality of free blocks and a second map containing a second set of nodes representing the addresses of and pointers to the plurality of free blocks in the first map. The allocator may insert a newly-freed block contained in a deletion request into the first map and the second map. The allocator may coalesce the plurality of free blocks in the first map after responding to deletion requests by combining an existing free block in the plurality of free blocks with by an adjacent newly-freed block.

[0017] Sharing of STL objects among processes as described herein has one or more of the following advantages. One advantage of this invention is its total portability between different versions of the STL because the invention does not utilize the STL at a source code level. Another advantage of this invention is that it is intuitive and easy to use, requiring no new methods or interfaces for the developer to learn. Yet another advantage of the invention is its ability to store any number of STL objects in one shared memory segment, with the shared memory instrumented to provide various usage details such as total size of the shared memory segment, total number of free blocks in the free store, and the granularity of the free blocks. Furthermore, this invention eliminates any type of same-machine sockets and COM usage, thus freeing valuable resources. As soon as the shared memory is mapped to the requesting process, access to the data is instantaneous. Finally, this invention offers the advantage of freeing the developers from the memory handling details, such as garbage collection.

[0018] The primary usage for this invention is for lookup data, where lookup data can be defined as an entity with a single writer and multiple readers. One process, typically an NT service, loads the lookup data and writes the data to the STL objects in shared memory. The data are available to the other processes once the processes load the shared memory.

[0019] The details of one or more embodiments of the invention are set forth in the accompanying drawings and the description below. Other features, objects, and advantages of the invention will be apparent from the description and drawings, and from the claims.

DESCRIPTION OF DRAWINGS

[0020] FIG. 1 illustrates a block system diagram for a shared memory segment common to three processes.

[0021] FIG. 2 illustrates an exemplary data structure diagram for an STL system map.

[0022] FIG. 3 is flowchart of a computer program implementation of the method of the invention.

[0023] FIG. 4A illustrates a data structure representing a free store map.

[0024] FIG. 4B illustrates a data structure representing the reverse free store map.

[0025] FIG. 5 is a flowchart detailing the allocator function's responses to allocation and deletion requests.

[0026] FIG. 6 is a detailed flowchart of a computer program implementation detailing the coalescing functionality of the allocator function.

[0027] FIG. 7 is a block diagram illustrating exemplary computer hardware components that constitute a suitable environment within which the principles of a shared memory system of the present invention may be implemented and operated.

[0028] Like reference numbers and designations in the various drawings indicate like elements.

DETAILED DESCRIPTION

[0029] STL objects are pointer-based data structures; they are conceptually understood as containers, or objects that contain other objects or contain pointers to objects. Containers at one level are data structures that manage a collection of elements and are responsible for the allocation and deallocation of those elements from memory. Containers typically have constructors and destructors along with operations for inserting and deleting elements. STL provides a variety of container types, include vectors, lists, dequeues, sets, maps, multimaps, queues, stacks, and the like. A simple container example is a vector, which is very much like an array with the ability to grow dynamically in size. A more complex type of container is the STL map, which is implemented as a balanced Red-Black Tree.

[0030] A Red-Black Tree is a tree with (key, value) pairs as the nodes; it is called "red-black" because every node is colored either red or black. STL maps represent a mapping from one type (the key type) to another type (the value type); the keys have to be unique, except for a multimap which allows duplicate keys, wherein each key may be associated with several values. In a STL map implemented as a Red-Black Tree, every node in the Tree has a value, and the value of any node is greater than the value of its left child, but less than the value of its right child. Every red node that is not a leaf has only black children. Every path from the root to a leaf contains the same number of black nodes, and the root node is black.

[0031] Generally, an STL object has two components: nodes and data. Nodes contain pointers to the previous and next nodes and any other accounting information like pointer to the data. In one implementation according to the technique of this invention, storage for both the nodes and the data of the STL object is allocated from the shared memory segment to ensure proper sharing--if storage is allocated only for the data from the shared memory segment, then other processes will not succeed in using the STL objects, as the nodes will be allocated memory by a process-specific private heap. STL objects have a default allocator to allocate memory from the heap. However, the design of the STL allows for a customized allocator, as explained below in connection with Fig. 5, to allocate memory from another location, such as a shared memory. This customized allocator tracks how much memory is used, how much memory is left in the shared memory segment, and the address for the next allocation, and the like.

[0032] Traditionally, data shared among a plurality of processes are stored in a file. While the traditional approach is adequate for simple object types such as integers and such, it is lacking in its ability to handle pointer-based data structures. STL objects, with its pointer-based data structures, constitute a good alternative to the traditional approach.

[0033] Referring to FIG. 1, the schematic diagram explains the layout of a multiprocessor system including the shared memory segment 130 common to three processes:

process 1 (100), process 2 (110), and process 3 (120). More than three processes can share the memory segment 130, but three processes are used here for illustrative purposes.

[0034] Process 1 (100) creates the named shared memory segment 130, which is of a fixed size of 10MB in one implementation. Process 1 (100) creates an STL system map in the shared memory 130 starting with a predetermine position such as byte 5 in section 140. This system map is an STL map designed to store the name and address of STL objects in the shared memory segment 130. The STL system map is anchored in the shared memory segment 130 starting at byte 5, and continues for the size of the map header determined at run-time.

[0035] The first four bytes constitute section 135 of the shared memory segment 130, representing the bytes for storing the next available address of the shared memory segment 130. For example, after the STL map is anchored at byte 5, the first four bytes of the shared memory segment 130 will now have a value of size of the map header, plus four, representing the next available address for any user STL object allocation. The data for the system map and the other user STL objects are stored in section 145 of the shared memory segment 130.

[0036] FIG. 2 shows a graphical representation of a data structure for system map 200. Every user STL object stored in system map 200 is represented as a row, with row heading 205 for key, and row heading 210 for value. For example, the first user STL object has name1 215 as key, at address1 220 as value. The second user STL object has name2 225, at address2 230. All user STL objects are listed as a series of rows until the last object in system map 200 is reached; this last user STL object has nameY 235 and addressY 240. Essentially, system map 200 is a mapping that uses as key the name of a user STL object and maps the name of the object to the address at which the object resides.

[0037] Referring to FIG. 3, the process of sharing data is explained by way of a flow chart. Initially, the shared memory segment 130 is created and identified with a name (step 300). After the system map 200 is anchored in shared memory segment 130 (step 305), a user STL object is created or received by, for example, Process 2 (step 310). All STL objects, in one implementation of the invention, are named objects. The system map 200 is examined to determine whether the name of this STL object already exists (step 315). If the name already exits, the associated address of the object is retrieved (step 320) from the system map 200, which stores the names and addresses of user STL objects in the shared memory 130. Furthermore, the object is pointed to the address associated with the name (step 325). If the name does not exist, the system calls an allocator that allocates storage from the shared memory segment for the user STL object. The allocator obtains the address for the next allocation by reading the first 4 bytes of the shared memory segment 130 (step 330). As previously stated, the first four bytes of the shared memory segment 130 always contain the address for the next allocation in one implementation. After obtaining the address, the same four bytes are updated with the new address for the next allocation (step 335). The new next allocation address is obtained by adding the original value and the current allocation size. In one implementation, this function ensures that no more than one process updates the address at the any given time.

[0038] Once an address is obtained, the name and the address pair of this previously unknown user STL object is inserted into the system map 200 (step 340). As the user adds data to the user STL object, the STL object requests storage from the same allocator, which in turn allocates memory from the shared memory 130.

[0039] Now both Process 1 and Process 2 refer to the same STL object at the same address. Different processes identify the same STL objects with the same address in the shared memory segment 130 by the same name. In other words, when the name is not found in the system map 200, an object is created and seated; if the name exists, the object is made to point to an already-existing location. This process is similar to the process through which NT handles kernel objects in different processes by giving them the same name.

[0040] The allocator also does garbage collection. Temporary objects are created when data is added to an STL object; these temporary objects need to be reclaimed once released to maximize storage space. Furthermore, some elements/nodes, such as an existing object, can be deleted by the user. The allocator does garbage collection by keeping track of released memory in two STL objects, a free store map 400 and a reverse free store map 401. The data structure of the two map objects are shown in Fig. 4A and Fig. 4B. The free store map 400 is a multimap that contains pairs of size (405) of the allocation and its address (410) of free blocks as the nodes. For example, in Fig. 4A there are four free blocks:

BLOCK 1: 31	876
BLOCK 2: 55	249
BLOCK 3: 55	981
BLOCK 4: 78	1076

a block of size 31 bytes (415) at address 876 (420), two blocks of size 55 bytes (425 and 435) at addresses 249 and 981 (430 and 440) and a block of size 78 bytes (445) at address 1076 (450). The key of the free store map 400 is size, or number of bytes (405). In other words, the map is sorted by size. For example. if an allocation request for 55 bytes comes in, the system will quickly locate BLOCK 2 with size 55 bytes (425) at address 249 (430) and deletes this entry from the free store map 400. Even though BLOCK 3 also satisfies the size limitation, the system, in one implementation, returns the first free block located in the free store map 400. This map 400 is useful in an allocation request where a free block with the right size can be quickly located. Once a free block of the right size is located for an allocation request, it needs to be deleted from the free store map 400. An STL multimap in general deletes all entries corresponding to the key. Therefore, in this case, both BLOCK 2 and BLOCK 3 will be deleted by standard STL operation, an undesirable result. In one implementation, the reverse free store map 401 is used in this delete operation ancillary to an allocation request to make sure only one free block in free store map 400 is deleted, as will be explained below. The reverse free store map 401 will also be used in an independent deletion request not associated with an allocation request, as explained below in conjunction with FIG. 6.

[0041] The reverse free store map 401 is shown in Fig. 4B. The use of the reverse free store map 401 is in the context of a deletion request, where the input parameter is the address (450) rather than size (405). Associated with the address 450 in the reverse free store map 401 is pointer 455 pointing back to free store map 400. The key of the reverse free map 401 is address 450. When an ancillary deletion request is encountered, the system searches reverse free store map 401 for a corresponding address and a pointer back to the free store map 401. The same four free blocks shown in FIG. 4A is repeated for FIG. 4B.

[0042] As stated above, when the suitable free block with 55 bytes (425) at address 249 (430) is to be deleted from the free store map 400, the system searches for a pointer from among pointers 460-475 that corresponds to address 249 (430) in the reverse free store map 401. Pointer 465 corresponds to address 249 (430), and points back to the right entry of size 55 (425) and address 249 (430), which will be allocated in response to an allocation request and needs to be deleted from the free store map 400.

[0043] Fig. 5 is a flow chart of how the allocator keeps track of requests for allocation and deletion. The input parameter for an allocation request is the size of storage requested. The function returns, in one implementation, an address for the first free block found in the free store map 400 of equal or greater size. If no existing free block in the free store map 400 of equal or greater size is found, the return value is the address of a new block of exactly the requested size from the shared memory segment 130. The allocator also keeps track of requests for deletion. The input parameter for a deletion request is the address of the block to be freed.

[0044] When a request for allocation comes in (step 505), it first checks in the free store map 400 for a block of an exact or larger size (step 510). If it exists, it is removed from the free store map 400 (step 515) and its address is returned to the caller (step 520). If this block is of larger size, the extra portion of the block is put back on the free store map 400. For example, the allocation request is for a size of 12 bytes. If the free store map 400 returns a block of 15 bytes at address 367, the extra 3 bytes of this block is returned to the free store map 400 at address 379 (367+12). If an equal or higher size is not found in the free store map 400, then shared memory segment 130 is checked for room(step 525). If space is found, memory will be allocated from the shared memory segment 130 (step 530). If there is not enough memory in the shared memory segment 130, NULL will be returned to the caller (step 535).

[0045] When a block of memory is freed via a deletion request (step 540), the allocator is again called. It returns (stores) the size and the address of a freed block to the free store map 400 (step 545). It also simultaneously coalesces the storage (step 550). Coalescing is the process by which any adjacent free blocks are combined to form larger free blocks. This process is explained by the flow chart of Fig. 6. There are four scenarios that could happen: (1) the current freed block is not adjacent to any existing free blocks. In this case, no coalescing takes place (step 620); (2) the current freed block is adjacent to an existing free block of lower address (step 600). The two are combined to form a larger free block (step 615); (3) the current freed block is adjacent to an existing free block of higher address (step 600). The two are combined to form a larger block (step 615); and (4) the current freed block falls exactly between two free blocks (step 605). In this case, the three free blocks are combined to form one free block (step 610). The coalescing calls for the use the free store map 400 and reverse free store map 401 again. While the address of the just-freed block is not in either map, a quick search is done to look for neighbors for coalescing as described above, returning a slot where the recently freed block should be inserted. After coalescing if any upper or lower neighbor is found, the coalesced block is inserted into free store map 400 and reverse free store map 401.

[0046] Another feature of the allocator is its ability to provide a snapshot of the current shared memory segment 130. By sending a special parameter to this function, it prints the total size of the shared memory segment 130, such as the size used, total number of free blocks in the free store map 400 and a distribution of allocation in terms of granularity i.e., how many blocks free with a size of 1-10 bytes, 11-20 bytes, 21-30 bytes, 31-40 bytes and more than 40 bytes.

[0047] The free store map 400 and the reverse free store map 401 are not stored in the shared memory segment 130; they are process-specific garbage collection mechanisms and are not available system-wide. In other words, any information regarding the free store map 400 available to one process is not visible to another process using the identical shared memory segment 130. If the processes are both writer and reader processes and each process has its own free store map and reverse free store map, there is a risk that certain pockets of available space in the shared memory 130 will be known to only one process. Therefore, the better implementation is to store the free store map 400 and the reverse store map 401 in a writer process only.

[0048] The two examples below illustrate two STL, objects, one map and one vector. They are created in one process (with 'W' option) and read in the second process (with 'R' option). In effect, two processes are created by compiling and running each example first with the 'W' option, and then with the 'R' option. The objects created in the 'W' process can be accessed in the 'R' process.

```
#pragma warning (disable: 4786 4503)
#include "IvpSTL.h"
using namespace std;

void main(int argc,char** argv)
{
        KeyValMap m("MapTest");

      if (!strcmp (argv[1], "w"))
          {
              CIvpWriteString key, val;

              key = "keyl";
              val = "valuel";
              m. insert (make_pair (key, val));

              key = "key2";
              val = "value2";
              m. insert (make_pair (key, val));

              key = "key3";
              val = "value3";
              m. insert (make_pair (key, val));

              //If the shared memory usage needs to be seen for any diagnostics
              IvpStd::ManageFreeSlot(0,0,IvpStd::STATS);
              getchar();
              return;
          }

//READ PORTION

        printf("in Read mode ...... \n");
        KeyValIter mapIt;
        CIvpReadString key("key2");
        mapIt = m.find(key);
        if   (mapIt != m. end())
                cout «
                    "Key Found. Key = " «
                    (*mapIt).first «
                    " Value = " «
                    (*mapIt).second «
                endl;
        else
                printf (“end\n”);
        return;
}
```

Example 2:

```
- - - - - - - - - - - -
#pragma warning (disable: 4786 4503)
#include "IvpSTL.h"
using namespace std;

struct XYZ  {
        int i;
        double d;
        CIvpWriteString s;
};

void main(int argc,char** argv)
{
        CIvpVector<XYZ> myVec("VectorTest");

     if (!strcmp (argv[1], "w"))
        {
            XYZ abc;

            abc.i = 12;
            abc.d = 34.67;
            abc.s = "First";

            myVec.push_back (abc);

            abc.i = 74;
            abc.d = 91.24;
            abc.s = "Second";

            myVec.push back (abc);

            getchar();
            return;
        }

//READ PORTION

        printf("in Read mode ...... \n");
        CIvpVector<XYZ>::VECDEF::iterator vecIt;
        for (vecIt = myVec.begin(); vecIt != myVec.end(); vecIt++)
            cout «  "i = [" « (*vecIt) .i « "] " «
                        "d = [" « (*vecIt) .d « "] " «
                            "s = [" « (*vecIt) .s « "] " « endl;

    return;
}
```

[0049] The above two examples can be combined to see two STL objects, one map and one vector, created in one process (with 'W' option) and read in the second process (with 'R' option).

[0050] Compile and run each example first with 'W' option and then with 'R' option. The STL object created in the first process can be accessed in the second process.

[0051] The thread-safety of STL objects in shared memory 130 is no different from that of objects residing in a process private heap. It is still the responsibility of the developer to ensure that different threads (processes) do not cause problems. In order to facilitate such a requirement which exists even without the shared memory 130, locking functionality can be added to the STL objects. The lock is, in one implementation, a Single Writer/Multi Reader Guard commonly known in the art.

[0052] Referring to Fig. 7 the invention may be implemented in digital electronic circuitry or in computer hardware, firmware, software, or in combinations of them. Apparatus of the invention may be implemented in a computer program product tangibly embodied in a machine-readable storage device for execution by a computer processor; and method steps of the invention may be performed by a computer processor executing a program to perform functions of the invention by operating on input data and generating output. Suitable processors include, by way of example, both general and special purpose microprocessors. Generally, a processor will receive instructions and data from a read-only memory 720 and/or a random access memory. Storage devices suitable for tangibly embodying computer program instructions include all forms of non-volatile memory, including by way of example semiconductor memory devices, such as EPROM, EEPROM, and flash memory devices; magnetic disks such as internal hard disks and removable disks 225; magneto-optical disks; and CD-ROM disks. Any of the foregoing may be supplemented by, or incorporated in, custom-designed ASICs (application-specific integrated circuit).

[0053] The essential elements of a computer are a processor 705 for executing instructions and a memory. A computer can generally also receive programs and data from a storage medium such as an internal disk (not shown) or a removable disk 725. These elements will be found in a conventional desktop 700 or workstation computer as well as other computers suitable for executing computer programs implementing the methods described here.

[0054] A number of embodiments of the invention have been described. Nevertheless, it will be understood that various modifications may be made without departing from the spirit and scope of the invention. Accordingly, other embodiments are within the scope of the following claims.